Drivetrain Library Documentation

Release 1.0

Brendan Doherty

Dec 11, 2019

Contents

A collection of motor drivers classes and specialized drivetrain classes to coordinate the motors' objects in generic configurations. This takes advatage of the threading module for smoothing motor input commands in background running threads. This was developed for & tested on the Raspberry PI. For running this library on CicuitPython devices (that don't have access to the threading module) like the Adafruit ItsyBitsy M4, we have added a fallback function called "sync()" that should get called at least once in the application's main loop.

CHAPTER 1

Dependencies

This library requires

- the `digitalio` and `pulseio` modules from the adafruit-blinka Library
- for serial communications: pyserial
- and for using the nRF24L01 as an interface: circuitpython-nrf24l01

CHAPTER 2

Installation

Currenty, there is no plan to deploy this library to pypi yet.

You can easily install this library to your Raspberry Pi in the terminal using the following commands:

```
git clone http://github.com/DVC-Viking-Robotics/Drivetrain.git
cd Drivetrain
python3 setup.py install
```

Some cases may require the last command be prefixed with `sudo` or appended with `--user`.

Installing this library should also automatically install the dependencies listed above (platform permitting).

CHAPTER 3

Examples

Try out any of the simple test examples in the examples to make sure everything (including pin connections & library installation) is setup correctly.

CHAPTER 4

Table of Contents

Note: To minimize the margin of error during these tests, we suggest connecting each motor pin (specified in the example codes) to its own resistor and LED (in series). By doing so, you can distinguish between pinout errors and faulty motor driver ICs more easily (not to mention doing away with the motors' isolated power requirements).

4.1 BiMotor test

Listing 1: examples/bimotor_test.py

```
"""
A simple test of the BiMotor class

This iterates through a list of motor commands
and prints the ellapsed time taken to acheive each command
"""
# pylint: disable=invalid-name
import time
import board
from drivetrain import BiMotor

motor = BiMotor([board.D22, board.D13], ramp_time=2000)
Value = [-25, 25, -100, 100, 0]
for test in Value:
    # send input instructions
    # NOTE we convert the percentage value to range [-65535, 65535]
    motor.cellerate(test * 655.35)
    start = time.monotonic()
    t = start
    # do a no delay wait for at most 3 seconds
    while motor.is_cellerating and t < start + 3:
```

(continues on next page)

(continued from previous page)

```
        t = time.monotonic()
    print('test result {} took {} seconds'.format(motor.value, t - start))
```

4.2 PhasedMotor test

Listing 2: examples/phasedmotor_test.py

```
"""
A simple test of the PhasedMotor class

This iterates through a list of motor commands
and prints the ellapsed time taken to acheive each command
"""
# pylint: disable=invalid-name
import time
import board
from drivetrain.motor import PhasedMotor

motor = PhasedMotor([board.D17, board.D18], ramp_time=2000)
Value = [-25, 25, -100, 100, 0]
for test in Value:
    # send input instructions
    # NOTE we convert the percentage value to range [-65535, 65535]
    motor.cellerate(test * 655.35)
    start = time.monotonic()
    t = start
    # do a no delay wait for at most 3 seconds
    while motor.is_cellerating and t < start + 3:
        t = time.monotonic()
    print('test result {} took {} seconds'.format(motor.value, t - start))
```

4.3 StepperMotor test

Listing 3: examples/steppermotor_test.py

```
"""A simple test of the StepperMotor class driving a 5V microstepper"""
# pylint: disable=invalid-name
import time
import board
from drivetrain.stepper import StepperMotor

motor = StepperMotor([board.D13, board.D12, board.D11, board.D10])
Steps = [-256, 256, 0] # 1024, 2048, 4096]
Angle = [-15, 15, 0] # 180, 360]
Value = [-25, 25, 0] # -50, 100, 0]
for test in Value:
    motor.value = test # send input instructions
    # do a no delay wait for at least 2 seconds
    start = time.monotonic()
    t = start
    end = None
```

(continues on next page)

(continued from previous page)

```
    while motor.is_cellerating or t < start + 2:
        t = time.monotonic()
        if not motor.is_cellerating and end is None:
            end = t
            print(repr(motor))
            print('value acheived in', end-start, 'seconds')
        # elif motor.is_cellerating:
        #     print(repr(motor))
```

4.4 Tank Drivetrain test

Listing 4: examples/tank_test.py

```
"""
A simple test of the Tank drivetrain class.

This iterates through a list of drivetrain commands
and tallies up the ellapsed time taken to acheive each set of commands
as well as the ellapsed time taken for each motor to acheive each individual command
"""
# pylint: disable=invalid-name
import time
import board
from drivetrain.drivetrain import Tank, BiMotor

mymotors = [BiMotor([board.D22, board.D13], ramp_time=2000),
            BiMotor([board.D17, board.D18], ramp_time=2000)]
d = Tank(mymotors)
testInput = [[100, 0],
             [-100, 0],
             [0, 0],
             [0, 100],
             [0, -100],
             [0, 0]]
for test in testInput:
    # use the list `end` to keep track of each motor's ellapsed time
    end = []
    # NOTE we convert a percentage to range of an 32 bit int
    for i, t_val in enumerate(test):
        test[i] = t_val * 655.35
    for m in mymotors:
        # end timer for motor[i] = end[i]
        end.append(None)
    d.go(test)  # send input commands
    # unanimous start of all timmers
    start = time.monotonic()
    t = start
    # do a no delay wait for at least 3 seconds
    while d.is_cellerating or t < start + 3:
        t = time.monotonic()
        for j, m in enumerate(mymotors):
            if not m.is_cellerating and end[j] is None:
                end[j] = t

```

(continues on next page)

(continued from previous page)

```
    print('test commands {} took {} seconds'.format(repr(test), t - start))
    for j, m in enumerate(mymotors):
        if end[j] is not None:
            print('motor {} acheived {} in {} seconds'.format(j, m.value, end[j]-
↪start))
        else:
            print("motor {} didn't finish cellerating and a has value of {}".format(j,
↪ m.value))
    print(' ') # for clearer print statement grouping
```

4.5 Automotive Drivetrain test

Listing 5: examples/automotive_test.py

```
"""
A simple test of the Automotive drivetrain class.

This iterates through a list of drivetrain commands
and tallies up the ellapsed time taken to acheive each set of commands
as well as the ellapsed time taken for each motor to acheive each individual command
"""
# pylint: disable=invalid-name
import time
import board
from drivetrain.drivetrain import Automotive, PhasedMotor

mymotors = [PhasedMotor([board.D22, board.D13], ramp_time=2000),
            PhasedMotor([board.D17, board.D18], ramp_time=2000)]
d = Automotive(mymotors)
testInput = [[100, 0],
             [-100, 0],
             [0, 0],
             [0, 100],
             [0, -100],
             [0, 0]]
for test in testInput:
    # use the list `end` to keep track of each motor's ellapsed time
    end = []
    # NOTE we convert a percentage to range of an 32 bit int
    for i, t_val in enumerate(test):
        test[i] = t_val * 655.35
    for m in mymotors:
        # end timer for motor[i] = end[i]
        end.append(None)
    d.go(test)  # send input commands
    # unanimous start of all timmers
    start = time.monotonic()
    t = start
    # do a no delay wait for at least 3 seconds
    while d.is_cellerating or t < start + 3:
        t = time.monotonic()
        for j, m in enumerate(mymotors):
            if not m.is_cellerating and end[j] is None:
                end[j] = t
```

(continues on next page)

(continued from previous page)

```

    print('test commands {} took {} seconds'.format(repr(test), t - start))
    for j, m in enumerate(mymotors):
        if end[j] is not None:
            print('motor {} acheived {} in {} seconds'.format(j, m.value, end[j]-
↪start))
        else:
            print("motor {} didn't finish cellerating and a has value of {}".format(j,
↪ m.value))
    print(' ') # for clearer print statement grouping
```

4.6 nRF24L01 receiving test

Listing 6: examples/nrf24l01_rx_test.py

```
"""
Example of library usage receiving commands via an
nRF24L01 transceiver to control a Mecanum drivetrain.
"""
import board
from digitalio import DigitalInOut as Dio
from circuitpython_nrf24l01 import RF24
from drivetrain import Mecanum, BiMotor, NRF24L01rx

# instantiate transceiver radio on the SPI bus
nrf = RF24(board.SPI(), Dio(board.D5), Dio(board.D4))

# instantiate motors for a Mecanum drivetrain in the following order
# Front-Right, Rear-Right, Rear-Left, Front-Left
motors = [
    BiMotor([board.RX, board.TX]),
    BiMotor([board.D13, board.D12]),
    BiMotor([board.D11, board.D10]),
    BiMotor([board.D2, board.D7])
    ]
# NOTE there are no more PWM pins available

# instantiate receiving object for a Mecanum drivetrain
d = NRF24L01rx(nrf, Mecanum(motors))

while True: # this runs forever
    d.sync()
# doing a keyboard interupt will most likely leave the SPI bus in an
# undesirable state. You must do a hard-reset of the circuitoython MCU to
# reset the SPI bus for continued use. This code assumes power is lost on exit.
```

4.7 Drivetrain Configurations

Important: Other motor libraries have implemented the "DC braking" concept in which all coils of the motor are energized to lock the rotor in place using simultaneously opposing electromagnetic forces. Unlike other motor libraries, we DO NOT assume your motors' driver circuit contains flyback diodes to protect its transistors (even though

they are practically required due to Lenz's Law). Therefore, passing a desired speed of `0` to any of the `cellerate()` or `go()` functions of the drivetrain and motor objects will effectively de-energize the coils in the motors.

4.7.1 Tank Drivetrain

class drivetrain.drivetrain.**Tank**(*motors*, *max_speed=100*)

A Drivetrain class meant to be used for motor configurations where propulsion and steering are shared tasks (also known as a "Differential" Drivetrain). For example: The military's tank vehicle essentially has 2 motors (1 on each side) where propulsion is done by both motors, and steering is controlled by varying the different motors' input commands.

Parameters

- **motors** (*list*) – A `list` of motors that are to be controlled in concert. Each item in this `list` represents a single motor object and must be of type *Solenoid*, *BiMotor*, *PhasedMotor*, or *StepperMotor*. The first 2 motors in this `list` are used to propell and steer respectively.
- **max_speed** (*int*) – The maximum speed as a percentage in range [0, 100] for the drivetrain's forward and backward motion. Defaults to 100%. This does not scale the motor speed's range, it just limits the top speed that the forward/backward motion can go.

go(*cmds*, *smooth=None*)

This function applies the user input to the motors' output according to drivetrain's motor configuration stated in the contructor documentation.

Parameters

- **cmds** (*list*) – A `list` of input motor commands to be processed and passed to the motors. This list must have at least 2 items (input values), and any additional items will be ignored. A `list` of length less than 2 will throw a `ValueError` exception.

  Important: Ordering of the motor inputs contained in this list/tuple matters. They should correspond to the following order:

  1. left/right magnitude in range [-65535, 65535]
  2. forward/reverse magnitude in range [-65535, 65535]

- **smooth** (*bool*) – This controls the motors' built-in algorithm that smooths input values over a period of time (in milliseconds) contained in the motors' *ramp_time* attribute. If this parameter is not specified, then the drivetrain's *smooth* attribute is used by default. This can be disabled per motor by setting the *ramp_time* attribute to `0`, thus the smoothing algorithm is automatically bypassed despite this parameter's value.

  Note: Assert this parameter (set as `True`) for robots with a rather high center of gravity or if some parts are poorly attached. The absence of properly smoothed acceleration/deceleration will likely make the robot fall over or loose parts become dislodged on sudden and drastic changes in speed.

  ---

is_cellerating

This attribute contains a `bool` indicating if the drivetrain's motors' speed is in the midst of changing. (read-only)

max_speed
This attribute determines a motor's top speed. Valid input values range [0, 100].

smooth
This attribute enables (`True`) or disables (`False`) the input smoothing alogrithms for all motors (solenoids excluded) in the drivetrain.

stop()
This function will stop all motion in the drivetrain's motors

sync()
This function should be used at least once per main loop iteration. It will trigger each motor's subsequent sync(), thus applying the smoothing input operations if needed. This is not needed if the smoothing algorithms are not utilized/necessary in the application

4.7.2 Automotive Drivetrain

class drivetrain.drivetrain.**Automotive**(*motors*, *max_speed=100*)
A Drivetrain class meant to be used for motor configurations where propulsion and steering are separate tasks. The first motor is used to steer, and the second motor is used to propell. An example of this would be any remote control toy vehicle.

Parameters

- **motors** (*list*) – A `list` of motors that are to be controlled in concert. Each item in this `list` represents a single motor object and must be of type *Solenoid* (steering only), *BiMotor*, *PhasedMotor*, or *StepperMotor*. The 2 motors in this `list` are used to steer and propell respectively.
- **max_speed** (*int*) – The maximum speed as a percentage in range [0, 100] for the drivetrain's forward and backward motion. Defaults to 100%. This does not scale the motor speed's range, it just limits the top speed that the forward/backward motion can go.

go(*cmds*, *smooth=None*)
This function applies the user input to motor output according to drivetrain's motor configuration.

Parameters

- **cmds** (*list*) – A `list` of input motor commands to be passed to the motors. This `list` must have at least 2 items (input values), and any additional item(s) will be ignored. A `list` of length less than 2 will throw a `ValueError` exception.

Important: Ordering of the motor inputs contained in this list/tuple matters. They should correspond to the following order:

1. left/right magnitude in range [-65535, 65535]
2. forward/reverse magnitude in range [-65535, 65535]

- **smooth** (*bool*) – This controls the motors' built-in algorithm that smooths input values over a period of time (in milliseconds) contained in the motors' *ramp_time* attribute. If this parameter is not specified, then the drivetrain's *smooth* attribute is used by default. This can be disabled per motor by setting the *ramp_time* attribute to `0`, thus the smoothing algorithm is automatically bypassed despite this parameter's value.

Note: Assert this parameter (set as `True`) for robots with a rather high center of gravity or if some parts are poorly attached. The absence of properly smoothed accelera-

tion/deceleration will likely make the robot fall over or loose parts become dislodged on sudden and drastic changes in speed.

---

is_cellerating
This attribute contains a `bool` indicating if the drivetrain's motors' speed is in the midst of changing. (read-only)

max_speed
This attribute determines a motor's top speed. Valid input values range [0, 100].

smooth
This attribute enables (`True`) or disables (`False`) the input smoothing alogrithms for all motors (solenoids excluded) in the drivetrain.

stop()
This function will stop all motion in the drivetrain's motors

sync()
This function should be used at least once per main loop iteration. It will trigger each motor's subsequent sync(), thus applying the smoothing input operations if needed. This is not needed if the smoothing algorithms are not utilized/necessary in the application

4.7.3 Locomotive Drivetrain

class drivetrain.drivetrain.**Locomotive**(*solenoids*, *switch*)
This class relies soley on one *Solenoid* object controlling 2 solenoids in tandem. Like with a locomotive train, applied force is alternated between the 2 solenoids using a boolean-ized pressure sensor or switch to determine when the applied force is alternated.

Parameters

- **solenoids** (`Solenoid`) – This object has 1 or 2 solenoids attached. It will be used to apply the force for propulsion.
- **switch** (*Pin*) – This should be the (`board` module's) `Pin` that is connected to the sensor that will be used to determine when the force for propulsion should be alternated between solenoids.

Note: There is no option to control the speed in this drivetrain class due to the nature of using solenoids for propulsion. Electronic solenoids apply either their full force or none at all. We currently are not supporting dynamic linear actuators (in which the force applied can vary) because they are basically motors simulating linear motion via a gear box controlling a shaft's extension/retraction. This may change when we support servos though.

---

stop()
This function stops the process of alternating applied force between the solenoids.

go(*forward*)
This function starts the process of alternating applied force between the solenoids with respect to the specified direction.

Parameters **forward** (*bool*) – `True` cylces the forces in a way that invokes a forward motion. `False` does the same but invokes a force in the backward direction.

Note: Since we are talking about applying linear force to a wheel or axle, the direction is entirely dependent on the physical orientation of the solenoids. In other words, the armature of one solenoid

should be attached to the wheel(s) or axle(s) in a position that is always opposite the position of the other solenoid's armature on the same wheel(s) or axel(s).

---

sync()
This function should be used at least once in the application's main loop. It will trigger the alternating of each solenoid's applied force. This IS needed on MCUs (microcontroller units) that can't use the threading module.

is_cellerating
This attribute contains a `bool` indicating if the drivetrain's applied force via solenoids is in the midst of alternating. (read-only)

4.7.4 Mecanum Drivetrain

class drivetrain.drivetrain.**Mecanum**(*motors*, *max_speed=100*)
A Drivetrain class meant for motor configurations that involve 4 motors for propulsion and steering are shared tasks (like having 2 Tank Drivetrains). Each motor drives a single mecanum wheel which allows for the ability to strafe.

Parameters

- **motors** (*list*) – A `list` of motors that are to be controlled in concert. Each item in this `list` represents a single motor object and must be of type *BiMotor*, *PhasedMotor*, or *StepperMotor*. The motors `list` should be ordered as follows:
  - Front-Right
  - Rear-Right
  - Rear-Left
  - Front-Left
- **max_speed** (*int*) – The maximum speed as a percentage in range [0, 100] for the drivetrain's forward and backward motion. Defaults to 100%. This does not scale the motor speed's range, it just limits the top speed that the forward/backward motion can go.

is_cellerating
This attribute contains a `bool` indicating if the drivetrain's motors' speed is in the midst of changing. (read-only)

max_speed
This attribute determines a motor's top speed. Valid input values range [0, 100].

smooth
This attribute enables (`True`) or disables (`False`) the input smoothing alogrithms for all motors (solenoids excluded) in the drivetrain.

stop()
This function will stop all motion in the drivetrain's motors

sync()
This function should be used at least once per main loop iteration. It will trigger each motor's subsequent sync(), thus applying the smoothing input operations if needed. This is not needed if the smoothing algorithms are not utilized/necessary in the application

go(*cmds*, *smooth=None*)
This function applies the user input to the motors' output according to drivetrain's motor configuration stated in the contructor documentation.

Parameters

- **cmds** (*list*) – A `list` of input motor commands to be processed and passed to the motors. This list must have at least 2 items (input values), and any additional items will be ignored. A `list` of length less than 2 will throw a `ValueError` exception.

  Important: Ordering of the motor inputs contained in this list/tuple matters. They should correspond to the following order:

  1. left/right magnitude in range [-65535, 65535]
  2. forward/reverse magnitude in range [-65535, 65535]
  3. strafe boolean. `True` uses the left/right magnituse as strafing speed. `False` uses the left/right magnitude for turning.

- **smooth** (*bool*) – This controls the motors' built-in algorithm that smooths input values over a period of time (in milliseconds) contained in the motors' *ramp_time* attribute. If this parameter is not specified, then the drivetrain's *smooth* attribute is used by default. This can be disabled per motor by setting the *ramp_time* attribute to `0`, thus the smoothing algorithm is automatically bypassed despite this parameter's value.

  Note: Assert this parameter (set as `True`) for robots with a rather high center of gravity or if some parts are poorly attached. The absence of properly smoothed acceleration/deceleration will likely make the robot fall over or loose parts become dislodged on sudden and drastic changes in speed.

4.8 Drivetrain Interfaces

4.8.1 NRF24L01

class drivetrain.interfaces.**NRF24L01**(*nrf24_object*, *address=b'rfpi0'*, *cmd_template='ll'*)

This class acts as a wrapper for circuitpython-nrf24l01 library for using a peripheral device with nRF24L01 radio transceivers. This is a base class to *NRF24L01tx* and *NRF24L01rx* classes.

Parameters

- **nrf24_object** (*RF24*) – The instantiated object of the nRF24L01 transceiver radio.
- **address** (*bytearray*) – This will be the RF address used to transmit/receive drivetrain commands via the nRF24L01 transceiver. For more information on this parameter's usage, please read the documentation on the using the `open_tx_pipe()`
- **cmd_template** (*str*) – This variable will be used as the "fmt" (Format String of Characters) parameter internally passed to the `struct.pack()` and `struct.unpack()` for transmiting and receiving drivetrain commands. The number of characters in this string must correspond to the number of commands in the `cmds` list passed to *go()*.

cmd_template

Use this attribute to change or check the format string used to pack or unpack drivetrain commands in `bytearray` form. Refer to Format String and Format Characters for allowed datatype aliases. The number of characters in this string must correspond to the number of commands in the `cmds` list passed to *go()*.

value
The most previous list of commands that were processed by the drivetrain object

address
This `bytearray` will be the RF address used to transmit/receive drivetrain commands via the nRF24L01 transceiver. For more information on this parameter's usage, please read the documentation on the using the `open_tx_pipe()`

class drivetrain.interfaces.**NRF24L01tx**(*nrf24_object*, *address=b'rfpi0'*, *cmd_template='ll'*)
Bases: *drivetrain.interfaces.NRF24L01*

This child class allows the remote controlling of an external drivetrain by transmitting commands to another MCU via the nRF24L01 transceiver. See also the *NRF24L01* base class for details about instantiation.

go(*cmds*)
Assembles a bytearray to be used for transmitting commands over the air to a receiving nRF24L01 transceiver.

Parameters **cmds** (*list*, *tuple*) – A `list` or `tuple` of `int` commands to be sent over the air using the nRF24L01. This `list`/`tuple` must have a length equal to the number of characters in the *cmd_template* string.

class drivetrain.interfaces.**NRF24L01rx**(*nrf24_object*, *drivetrain*, *address=b'rfpi0'*, *cmd_template='ll'*)
Bases: *drivetrain.interfaces.NRF24L01*

This child class allows the external remote controlling of an internal drivetrain by receiving commands from another MCU via the nRF24L01 transceiver.

Parameters **drivetrain** (`Tank`, `Automotive`, `Locomotive`) – The pre-instantiated drivetrain configuration object that is to be controlled.

See also the *NRF24L01* base class for details about instantiation.

sync()
Checks if there are new commands waiting in the nRF24L01's RX FIFO buffer to be processed by the drivetrain object (passed to the constructor upon instantiation). Any data that is waiting to be received is interpreted and passed to the drivetrain object.

go(*cmds*)
Assembles a list of drivetrain commands from the received bytearray via the nRF24L01 transceiver.

Parameters **cmds** (*list*, *tuple*) – A `list` or `tuple` of `int` commands to be sent the drivetrain object (passed to the constructor upon instantiation). This `list`/`tuple` must have a length equal to the number of characters in the *cmd_template* string.

4.8.2 USB

class drivetrain.interfaces.**USB**(*serial_object*, *cmd_template='ll'*)
This base class acts as a wrapper to pyserial module for communicating to an external USB serial device. Specifically designed for an Arduino running custom code.

Parameters

- **serial_object** (*busio.UART*, *serial.Serial*, *machine.UART*) – The instantiated serial object to be used for the serial connection.
- **cmd_template** (*str*) – This variable will be used as the "fmt" (Format String of Characters) parameter internally passed to the `struct.pack()` and `struct.unpack()` for transmiting and receiving drivetrain commands. The number of characters in this string must correspond to the number of commands in the `cmds` list passed to *go()*.

cmd_template
Use this str attribute to change or check the format string used to pack or unpack drivetrain commands in bytearray form. Refer to Format String and Format Characters for allowed datatype aliases. The number of characters in this string must correspond to the number of commands in the cmds list passed to *go()*.

value
The most previous list of commands that were processed by the drivetrain object

class drivetrain.interfaces.**USBtx**(*serial_object*, *cmd_template='ll'*)
Bases: *drivetrain.interfaces.USB*

This child class allows the remote controlling of an external drivetrain by transmitting commands to another MCU via USB serial connection. See also the *USB* base class for details about instantiation.

go(*cmds*)
Assembles a bytearray for outputting over the Serial connection.

Parameters **cmds** (*list*, *tuple*) – A list or tuple of int commands to be sent over the Serial connection. This list/tuple must have a length equal to the number of characters in the *cmd_template* string.

class drivetrain.interfaces.**USBrx**(*drivetrain*, *serial_object*, *cmd_template='ll'*)
Bases: *drivetrain.interfaces.USB*

This child class allows the remote controlling of an external drivetrain by receiving commands from another MCU via USB serial connection.

Parameters **drivetrain** (Tank, Automotive, Locomotive) – The pre-instantiated drivetrain configuration object that is to be controlled.

See also the *USB* base class for details about instantiation.

sync()
Checks if there are new commands waiting in the USB serial device's input stream to be processed by the drivetrain object (passed to the constructor upon instantiation). Any data that is waiting to be received is interpreted and passed to the drivetrain object.

go(*cmds*)
Assembles a list of drivetrain commands from the received bytearray over the USB serial connection.

Parameters **cmds** (*list*, *tuple*) – A list or tuple of int commands to be sent the drivetrain object (passed to the constructor upon instantiation). This list/tuple must have a length equal to the number of characters in the *cmd_template* string.

4.9 Motor Types

4.9.1 Solenoid

class drivetrain.motor.**Solenoid**(*pins*, *ramp_time=0*)
This base class is meant be used as a parent to *BiMotor* and *PhasedMotor* classes of this module, but can be used for solenoids if needed. Solenoids, by nature, cannot be controlled dynamically (cannot be any value other than True or False). Despite the fact that this class holds all the smoothing input algorithms for its child classes, the output values, when instantiated objects with this base class, are not actually smoothed. With that said, this class can be used to control up to 2 solenoids (see also *value* attribute for more details) as in the case of an actual locomotive train.

Parameters

- **pins** (*list*) – A `list` of (`board` module's) `Pin` numbers that are used to drive the solenoid(s). The length of this `list` must be in range [1, 2] (any additional items/pins will be ignored).
- **ramp_time** (*int*) – This parameter is really a placeholder for the child classes *BiMotor* & *PhasedMotor* as it has no affect on objects instantiated with this base class. Changing this value has not been tested and will probably slightly delay the solenoid(s) outputs.

value
This attribute contains the current output value of the solenoid(s) in range [-1, 1]. An invalid input value will be clamped to an `int` in the proper range.

Note: Because this class is built to handle 2 pins (passed in the `pins` parameter to the constructor) and tailored for solenoids, any negative value will only energize the solenoid driven by the second pin . Any positive value will only energize the solenoid driven by the first pin. Alternatively, a `0` value will de-energize both solenoids.

4.9.2 BiMotor

class drivetrain.motor.**BiMotor**(*pins*, *ramp_time=500*)
This class is meant be used for motors driven by driver boards/ICs that expect 2 PWM outputs . Each pin represent the controlling signal for the motor's speed in a single rotational direction.

Parameters

- **pins** (*list*) – A `list` of (`board` module's) `Pin` numbers that are used to drive the motor. The length of this `list` or `tuple` must be in range [1, 2]; any additional items/pins will be ignored, and a `ValueError` exception is thrown if no pins are passed (an empty `tuple`/`list`). If only 1 pin is passed, then the motor will only rotate in 1 direction depending on how the motor is connected to the motor driver.
- **ramp_time** (*int*) – The time (in milliseconds) that is used to smooth the motor's input. Default is 500. This time represents the maximum amount of time that the input will be smoothed. Since the change in speed is also used to determine how much time will be used to smooth the input, this parameter's value will represent the time it takes for the motor to go from full reverse to full forward and vice versa. If the motor is going from rest to either full reverse or full forward, then the time it takes to do that will be half of this parameter's value. This can be changed at any time by changing the *ramp_time* attribute.

cellerate(*target_speed*)
A function to smoothly accelerate/decelerate the motor to a specified target speed.

Parameters **target_speed** (*int*) – The desired target speed in range of [-65535, 65535]. Any invalid inputs will be clamped to an `int` value in the proper range.

is_cellerating
This attribute contains a `bool` indicating if the motor's speed is in the midst of changing. (read-only)

ramp_time
This attribute is the maximum amount of time (in milliseconds) used to smooth the input values. A negative value will be used as a positive number. Set this to `0` to disable all smoothing on the motor input values or just set the *value* atribute directly to bypass the smoothing algorithm.

Note: Since the change in speed (target - initial) is also used to determine how much time will be used to smooth the input, this attribute's value will represent the maximum time it takes for the motor to go

from full reverse to full forward and vice versa. If the motor is going from rest to either full reverse or full forward, then the time it takes to do that will be half of this attribute's value.

sync()
This function should be used at least in the application's main loop iteration. It will trigger the smoothing input operations on the output value if needed. This is not needed if the smoothing algorithms are not utilized/necessary in the application

value
This attribute contains the current output value of the solenoid(s) in range [-65535, 65535]. An invalid input value will be clamped to an `int` in the proper range. A negative value represents the motor's speed in reverse rotation. A positive value reprsents the motor's speed in forward rotation.

4.9.3 PhasedMotor

class drivetrain.motor.**PhasedMotor**(*pins*, *ramp_time=500*)
This class is meant be used for motors driven by driver boards/ICs that expect:

- 1 PWM output (to control the motor's speed)
- 1 digital output (to control the motor's rotational direction)

Parameters

- **pins** (*list*) – A `list` of (`board` module's) `Pin` numbers that are used to drive the motor. The length of this `list`/`tuple` must be 2, otherwise a `ValueError` exception is thrown.

  Note: The first pin in the `tuple`/`list` is used for the digital output signal that signifies the motor's rotational direction. The second pin is used for PWM output that signifies the motor's speed.

- **ramp_time** (*int*) – The time (in milliseconds) that is used to smooth the motor's input. Default is 500. This time represents the maximum amount of time that the input will be smoothed. Since the change in speed is also used to determine how much time will be used to smooth the input, this parameter's value will represent the time it takes for the motor to go from full reverse to full forward and vice versa. If the motor is going from rest to either full reverse or full forward, then the time it takes to do that will be half of this parameter's value. This can be changed at any time by changing the *ramp_time* attribute.

cellerate(*target_speed*)
A function to smoothly accelerate/decelerate the motor to a specified target speed.

Parameters **target_speed** (*int*) – The desired target speed in range of [-65535, 65535]. Any invalid inputs will be clamped to an `int` value in the proper range.

is_cellerating
This attribute contains a `bool` indicating if the motor's speed is in the midst of changing. (read-only)

ramp_time
This attribute is the maximum amount of time (in milliseconds) used to smooth the input values. A negative value will be used as a positive number. Set this to `0` to disable all smoothing on the motor input values or just set the *value* atribute directly to bypass the smoothing algorithm.

Note: Since the change in speed (target - initial) is also used to determine how much time will be used to smooth the input, this attribute's value will represent the maximum time it takes for the motor to go from full reverse to full forward and vice versa. If the motor is going from rest to either full reverse or full forward, then the time it takes to do that will be half of this attribute's value.

sync()
This function should be used at least in the application's main loop iteration. It will trigger the smoothing input operations on the output value if needed. This is not needed if the smoothing algorithms are not utilized/necessary in the application

value
This attribute contains the current output value of the solenoid(s) in range [-65535, 65535]. An invalid input value will be clamped to an `int` in the proper range. A negative value represents the motor's speed in reverse rotation. A positive value reprsents the motor's speed in forward rotation.

4.9.4 StepperMotor

class drivetrain.stepper.**StepperMotor**(*pins*, *steps_per_rev=4096*, *degree_per_step=0.087890625*, *step_type='half'*, *rpm=60*)
A class designed to control unipolar or bipolar stepper motors. It is still a work in progress as there is no smoothing algorithm nor limited maximum speed applied to the motor's input.

Parameters

- **pins** (*list*, *tuple*) – A `list` or `tuple` of (`board` module) pins that are used to drive the stepper motor. The length of this `list` or `tuple` must be divisible by 2, otherwise a `ValueError` exception is thrown.
- **steps_per_rev** (*int*) – An `int` that represents how many steps it takes to complete a whole revolution. Defaults to 4096. This should correlate with information found in your motor's datasheet.
- **degree_per_step** (*int*, *float*) – The value that represents how many degrees the motor moves per single step. Defaults to 45/512 or 5.625°/64. This should correlate with information found in your motor's datasheet.
- **step_type** (*string*) – This parameter is used upon instantiation to specify what kind of stepping pattern the motor uses. Valid values are limited to:
  - `half` (default value)
  - `full`
  - `wave`

  This should correlate with information found in your motor's datasheet.
- **rpm** (*int*, *float*) – The maximum amount of rotations per minute. This should correlate with information found in your motor's datasheet.

is_cellerating
This attribute contains a `bool` indicating if the motor is in the midst of moving. (read-only)

stop()
Use this function when you want to abort any motion from the motor.

sync()
This function should be used only once per main loop iteration. It will trigger stepping operations on the motor if needed.

reset0angle()
A calibrating function that will reset the motor's zero angle to its current position. This function is also called when the motor's *value*, *steps*, or *angle* attributes are set to None. Additionally, this function will stop all movement in the motor.

rpm
This int attribute contains the maximum Rotations Per Minute and can be changed at any time.

angle
Represents the number of the motor's angle from its zero angle position with respect to the steps_per_rev parameter passed to the constructor. This value will be in range [-180, 180]. Input values can be any int or float as any overflow outside the range [0, 360] is handled accordingly.

steps
Represents the number of the motor's steps from its zero angle position with respect to the steps_per_rev parameter passed to the constructor. This value will be in range [steps_per_rev / -2, steps_per_rev / 2]. Input values can be any int as any overflow outside the range [0, steps_per_rev] is handled accordingly.

value
Represents the percentual value of the motor's angle in range [-100, 100] with respect to the steps_per_rev parameter passed to the constructor. Invalid input values will be constrained to an int in the range [-100, 100].

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

T

U

V